EUROGRAPHICS 2009 / P. Dutré and M. Stamminger
(Guest Editors)

Volume 28 (2009), Number 2

Table-based Alpha Compression

Per Wennersten and Jacob Ström

Ericsson Research

Abstract

In this paper we investigate low-bitrate compression of scalar textures such as alpha maps, down to one or two bits per pixel. We present two new techniques for 4 × 4 blocks, based on the idea from ETC to use index tables. We demonstrate that although the visual quality of the alpha maps is greatly reduced at these low bit rates, the quality of the final rendered images appears to be sufficient for a wide range of applications, thus allowing bandwidth savings of up to 75%. The 2 bpp version improves PSNR with over 2 dB compared to BTC at the same bit rate. The 1 bpp version is, to the best of our knowledge, the first public 1 bpp texture compression algorithm, which makes comparison hard. However, compared to just DXT5-compressing a subsampled texture, our 1 bpp technique improves PSNR with over 2 dB. Finally, we show that some aspects of the presented algorithms are also useful for the more common bit rate of four bits per pixel, achieving PSNR scores around 1 dB better than DXT5, over a set of test images.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.7]: Texture—

1. Introduction

As increases in processing power continue to outpace increases in memory bandwidth [Owe05], available bandwidth often becomes a performance-limiting factor in modern rasterizer architectures for computer games on consumer-level graphics cards in PCs [AMN03]. For mobile architectures, bandwidth becomes even more critical, as it affects power consumption and thereby battery life [AMS03]. One way of reducing bandwidth usage is to employ *texture compression*, pioneered by Knittel et al. [KSKS96], Beers et al. [BAC96] and Torborg and Kajiya [TK96]. The idea is to compress the textures and transfer them over the bus in compressed form, thus saving bandwidth. Compressed textures not only generate less bandwidth during rendering, but also demand less storage space, thus allowing for higher resolution textures for the same amount of texture memory. Our contribution is to show how table based compression, based on the intensity representation in ETC/iPACKMAN [SAM05] can be used to an advantage for compression of scalar 8-bit textures such as transparency (alpha) maps.

2. Previous work

Delp and Mitchell [DM79] propose a fixed rate image compression algorithm for gray scale images. Each pixel in a 4×4 image block can choose from two gray values using a bitmask. The bitmask and the two 8-bit grey values are stored explicitly in the block, yielding 2 bits per pixel (bpp). The scheme is extended to color by Campbell et al. [CDF*86] by using two colors instead of two gray levels. However, the limitation of only having two colors/graylevels per block gives rise to banding artifacts. This problem is greatly reduced by Iourcha et al. [INH99] by the introduction of two more, interpolated, colors to choose from. The algorithm called DXTC/S3TC compresses textures to 4 bits per pixel and is now the de facto standard for RGB texture compression on desktops. The DXT1 version of the standard includes support for RGBA textures with one bit alpha ("punch-through alpha"). This is done by replacing one of the interpolated values with black with alpha 0.0, whereas the other three values have alpha 1.0. A similar scheme for punch-through alpha RGBA is also used in the PVR-TC compression format by Fenney [Fen03]. However, PVR-TC includes an additional way of obtaining alpha which allows for smoother alpha transitions: PVR-TC works by blending two low-frequency signals, and it is possible to specify these signals using RGBA4443 instead of the regular RGB555, allowing greater control of the alpha signal at the expense of the color representation. DXTC has chosen a different path, using 64 bits for the RGB channels of the 4×4 texture block and another 64 bits for the alpha channel. This decoupling of the alpha from the color has some drawbacks — the encoding cannot take advantage of any correlation between the color and the transparency channels, nor can it exploit the fact that high quality color is mostly needed during opacity. However, one benefit of the decoupling is that the alpha compressor can be used as a general 8 bit data compressor; hence it has been used for luminance alpha textures as well

Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

as normal map textures using ATI's 3Dc [ATI05], and with the advent of programmable shaders the importance of such general data compression seems destined to increase.

Whereas the RGB part is coded similarly in the different DXTC codecs, the alpha channel is encoded using two different modes. The first mode, used in DXT2 and DXT3, simply stores the four most significant bits of the alpha value for each pixel. When decompressing, the most significant bits are then copied into the least significant bits, ensuring that both 0 and 255 can be represented. The difference between DXT2 and DXT3 is that the former encodes colors that are premultiplied by alpha, whereas the latter does not.

The second alpha compression mode is used in DXT4 and DXT5, and is a bit more complex. Two 8-bit alpha values are stored for each 4×4 block of pixels, and six new values are interpolated between them, for a total of eight possible alpha values per block. Three bits for each pixel in the block are then used to select one of these eight values. The procedure is illustrated in Figure 1. This technique thereby manages to avoid any banding in blocks with slowly varying alpha values, while keeping decent quality in blocks with a wider range of values. An improvement to the method is

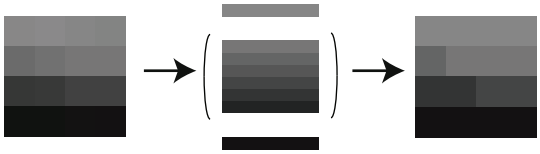

Figure 1: *An illustration of the alpha compression algorithm used in DXT4 and DXT5. Two colors are stored for each block, six colors are interpolated between them, and these eight colors are used when compressing the block.*

made by realizing that the interpolated values are the same regardless of the ordering of the initial 8-bit values: interpolating between 50 and 150 gives the same values as interpolating between 150 and 50. Therefore, the case where the second value is smaller than the first can be used to signal a different compression mode. In this new mode, the alpha values 0 and 255 are always among the eight possible values, and only four values are interpolated in-between the two supplied ones. This leads to improved performance for blocks where some pixels are either fully transparent or fully opaque, a fairly common case when storing opacity information in the channel. DXT4 differs from DXT5 only in the way it handles premultiplication. Therefore, in the following we will only refer to DXT5 although the arguments hold also for DXT4.

Our approach builds upon the ETC/iPACKMAN scheme, and therefore we will go through that in some detail. In ETC [SAM05] and its predecessor PACKMAN [SAM04], each pixel in a 4×2 pixel area can choose between four different *paint colors*. These paint colors are obtained by specifying a *base color* and then modifying the intensity of this base color using *offset values* from a table. For instance, if the base color is $(221, 132, 132)$ and the table of offset values is $\{-10, -5, 5, 10\}$, the resulting paint colors are $(211, 122, 122), (216, 127, 127), (226, 137, 137)$ and $(231, 142, 142)$. A two-bit *pixel index* is used for each pixel to choose between the four paint colors. Blocks containing smooth data will need fine variations, and for such blocks a table such as $\{-4, -2, 2, 4\}$ may be appropriate. However, for blocks with large variations, a table such as $\{-70, -28, 28, 70\}$ may be better. In ETC/PACKMAN this is solved by letting each 4×2 pixel block choose a table from a codebook of several tables. Smooth blocks then select "small" tables, and high-contrast blocks can use "large" tables. This is similar to the DXT5-way of storing the min- and max value, but is more efficient in terms of compression, as we will see.

3. Low-Bitrate Compression

One interesting aspect of DXTC is that both compression modes with a full alpha channel use four bits per pixel for alpha, and four bits per pixel for color. This means that the information in the alpha channel is only compressed down to 50% of its original size, with very little loss of quality as a result. In contrast, the color channels are compressed down to one sixth of their original size, resulting in a noticeable reduction in quality. Whether this difference in quality is desirable will of course depend on the application, and exactly how the channels are used in the pixel shader, but it seems reasonable to believe that further compression of the alpha channel might be acceptable for some applications. Because of this, we have developed algorithms for compressing an alpha map down to two bits or one bit per pixel, 32 or 16 bits per 4×4 block, and investigated the effects this has on the rendered image for a few different types of alpha maps.

3.1. Description of the 32-bit Algorithm

Our proposed 2 bpp scheme starts by defining three *paint alpha values* similar to the four paint colors used in ETC. First, a *base alpha value* is supplied. Only the four most significant bits are stored, and during decompression, these are copied into the least significant bits to get the complete 8-bit base alpha value. Next, a codebook of 16 tables with three offset values each is stored on chip, and each block supplies an index into this codebook, selecting one of the tables of three offsets. Finally, each of the three paint alpha values are created by adding one of the three offsets to the base value and clamping the sum to eight bits. This procedure is shown in Figure 2, along with the codebook of 16 offset tables we used. Next, ETC and DXT5 both proceed by selecting one of the available colors for each pixel in a block. In our case, however, we only have 24 bits remaining for the 16 pixels, which is not enough to choose between the three values. Our

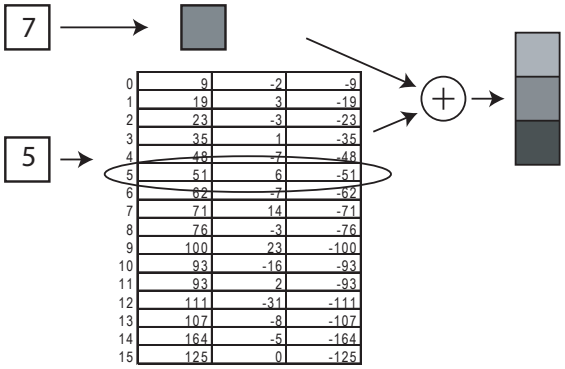

Figure 2: *An illustration of the technique of using a base value and an offset table index to determine possible alpha values for a block. The index selects an offset table to be used, and the base value is then added to each value in the table which after clamping gives us the three final alpha values for the block.*

solution is to divide the pixels into groups of two (horizontal neighbors in our case), each group using three bits to select between combinations of alpha values for the group. Because there are nine possible alpha value combinations and we can only select eight different combinations using three bits, one possible combination is discarded, and will never be used. We have discarded the combination with the lowest possible alpha value in one pixel followed by the highest value in the next, on the rationale that this combination should be one of the rarest due to inter-pixel correlation.

The 16 tables were obtained from training; starting with random values and successively changing the values while compressing a set of training images to see if the changes were beneficial. The training images were not used in the test set later used to evaluate the performance.

3.2. Description of the 16-bit Algorithm

In our 16-bit version, we use four bits to select a base alpha value and another four to select a table of offsets, just as in the 32-bit version. This leaves only eight bits for the pixel indices. Thus the approach taken in the previous scheme for encoding the pixel indices is no longer usable. However, eight bits are just enough to specify a line segment in a 4×4 block by encoding the start- and stop coordinates (x_1, y_1) and (x_2, y_2) using two bits per coordinate. We thus tried to use a line segment as the skeleton for our pixel indices, and found it to provide a rather fair trade-off between being able to specify small individual details (such as a single pixel) and larger connected components. The layout for the compressed block is shown in Figure 3. A line, one pixel wide, is drawn between the two pixels, as show in the "drawn line"-diagram in Figure 4.

Figure 3: *The bit layout of our 16-bit compressed block.*

index	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	-141	-54	-48	7	47	105	128	142
1	76	77	55	9	-29	-79	-100	-117
2	-109	-74	-63	-5	-14	36	35	28
3	75	7	-19	37	-42	-65	-58	-75
4	-60	-31	-25	-30	27	17	78	21
5	50	47	31	12	-4	-46	-114	-64
6	-49	-46	-41	2	4	23	145	91
7	2	22	60	41	15	-14	-24	-42
8	-42	-66	-41	-47	-14	-7	4	50
9	29	37	8	7	0	-47	-20	-31
10	-25	-19	-15	-12	-2	12	14	5
11	-6	-7	18	12	-12	43	-15	-18
12	-27	-29	-5	13	11	117	86	177
13	1	-2	10	-5	-6	-55	-9	-10
14	49	18	-16	32	20	3	-19	-6
15	50	42	36	13	-11	-13	12	62

Table 1: *offsets m_0 to m_7 for the 16 offset tables for 1 bpp compression. Every other table has been flipped to account for the case when both line endpoints lie in the same point. In all other cases, all 16 tables can either be flipped or used as-is.*

Variations in pixel indices are often smooth, and we have therefore devised a neighbor counting scheme to get smooth pixel indices from the line segment: Once the line has been drawn, a count is made for each of the remaining pixels, starting at 0. One is added for each diagonally neighboring pixel lying on the line, two is added for each vertically or horizontally neighboring pixel on the line, and finally one is added for each edge of the block neighboring the pixel if any neighbor pixels were on the line. This count then becomes the pixel index for the pixel, ranging from 0 to 7 with 7 reserved for pixels directly on the line. Thus it is possible to use eight values per table instead of four as in the 32-bit version. The procedure is shown in Figure 4. For instance, the pixel marked with "2" in the "pixel indices" diagram gets one point for having a diagonal neighbor on the line, and another point for also being an edge pixel, for a total count of two. The 16 possible offset tables used in this mode are shown in Table 1. Just as for the 32-bit version, these offset tables were found by training on data that was not part of the evaluation set.

With this method, results will be very similar if the positions of the first and second given pixels are swapped. We exploit this by reversing the order of the eight alpha values if the first given pixel lies after the second given pixel assuming a fixed ordering of the 16 pixels in the block (We use left-to-right, top-to-bottom). Note that this trick cannot be used

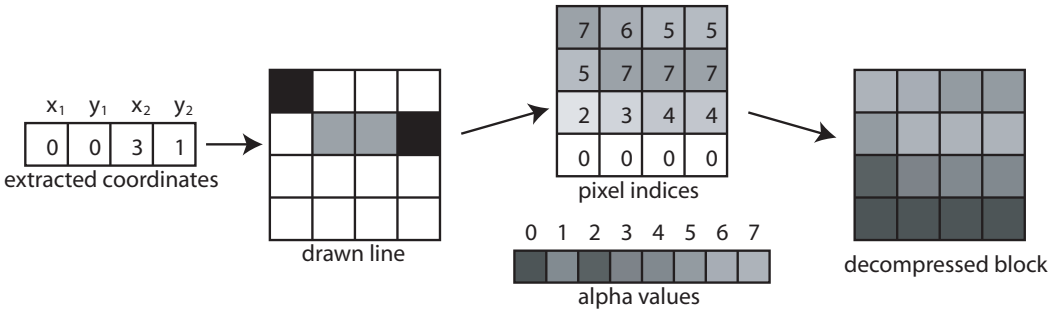

Figure 4: *An illustration of the proposed 16-bit compression algorithm. The eight alpha values are determined from the base value and offset index as in Figure 2. Each pixel selects one of these values based on the drawn line, resulting in the final decompressed block.*

when both endpoints of the line lie on the same pixel, and so the original offset tables are alternatively low to high alpha values or high to low. This way, blocks with line segments consisting of a single pixel still can reverse the ordering by selecting a slightly different table.

One drawback with the algorithm is that constant blocks cannot be represented, which will result in pattern-like artifacts in flat areas. Fortunately, such blocks do not contain much information and can therefore be very well compressed by a special mode that we implemented: A particular combination of $(x_1,y_1)(x_2,y_2)$-values is disallowed (we used $(0,0)(0,1)$) and such a block is instead decoded by filling all pixels with the eight first bits from the bit sequence (normally occupied by "base" and "code word"). This fix removes the artifacts, and also increases compression efficiency, since this type of block is very common.

Since compression to this format is intended to be performed offline before the texture is used, performance is not critical. Our compressor works by precomputing the 2^{16} possible final blocks, along with their average alpha value. Each block is then compressed by comparing the block to all possible compressed blocks and selecting the best one. The average alpha value for the blocks is used as an early reject method: if the squared difference between the averages of the block being compressed and a possible compressed block is larger than the lowest mean squared error we have obtained thus far, the possible block can be skipped without examining the individual alpha values.

3.3. Testing Method

There are three important properties that are desired of our low-bitrate algorithms. The first is that they should be reasonably efficient compared to other methods with similar bit rates, the second that the appearance of the final rendered scene using our compressed textures is acceptable, and the third that a hardware implementation of the decoder is reasonably efficient.

In order to test the first property we have performed comparisons with other methods achieving the same level of compression. For the 32-bit case, we have designed a DXTC-like compressor for comparison, which, instead of selecting a base value and an offset table, selects two base values and interpolates between them to obtain the third.We exploit the fact that the order in which the two alpha values are supplied is irrelevant, by having the interpolated value be the sum of $\frac{2}{3}$ of the first value and $\frac{1}{3}$ of the second. We have also implemented BTC [DM79] and included it in our comparison. For the 16-bit case we have compared the described algorithm with two different methods. The first, and simplest, is to use a lower resolution DXT5-encoded alpha texture, with half the horizontal and vertical resolution. This approach simply lowers the sampling rate in order to reach the bit rate target of 1 bpp, and as such represents a lower bound for the compression efficiency (quality per bit) we want. Secondly, as we are targeting a compressed size of just 16 bits, we have tried a brute-force approach of using vector quantization to optimize the set of 2^{16} possible compressed blocks. Vector quantization would not be practical for texture compression, as it would require storing 2^{16} blocks on-chip, equivalent to a megabyte of ROM, but is included to provide an indication of what the upper bound for compression efficiency might be in our case. Since the performance of vector quantization systems typically improve with increased training data, and since we do not have much training data available, we run the risk of getting too low an estimate of the upper bound. We therefore perform training directly on the test data: This gives our vector quantization implementation an unfair advantage that, if our training algorithm is efficient enough, should more than compensate for the lack of training data, hence providing a more reasonable upper bound estimate. For our proposed systems, the tables are obtained by training on data. In those cases, the training data is different from the test data. The three compression methods were compared on a set of 64 images, with mip-levels ranging from 512×512 to 8×8 pixels. For all

algorithms, including DXT5, BTC and the DXTC-like algorithms used for comparison, we use exhaustive search, which results in optimal compression.

In order to test the appearance of a rendered scene using our compression methods, we have selected a few different types of alpha textures for testing: one alpha map, one specular map and one parallax map. They are all from the 2006 computer game Oblivion, by Bethesda Softworks. They are originally compressed using DXT5, and we have further compressed them using our low bit rate algorithms, before decompressing them and using them in uncompressed form in the game to study the results. The third property, efficient hardware implementation, will be treated in Section 5.

3.4. Results

The results from our 32-bit and 16-bit comparisons are shown in Figures 5 and 6, respectively, where the PSNR score is plotted for different mip-sizes. PSNR is defined as $10\log_{10}(255^2/MSE_{set})$, where MSE_{set} is the mean square error over the set of textures to be measured (for instance a certain mipmap level) and is calculated as $MSE_{set} = (1/N)\sum_N MSE_{tex}$. MSE_{tex} is the mean squared error of an individual texture $MSE_{tex} = (1/(WH))\sum_{W,H}(a-\hat{a})^2$, W and H being the width and height of the texture respectively, and where a and $\hat{a}$ represent the original and compressed pixels. It is important to calculate an aggregated PSNR score this way, since the alternative way, to simply average the PSNR scores of the individual textures, will overstate the PSNR measure. For instance, a single texture with zero error will produce a PSNR score of infinity no matter what the errors are in the rest of the textures. By averaging the MSE_{tex} values instead, this problem is avoided. A higher PSNR value indicates a smaller error, and therefore increased quality.

Note in particular in the 16-bit comparison that further compressing the textures leads to significantly higher PSNR scores than simply lowering the resolution to achieve the same bit rate. We are also closer to the vector quantization algorithm, which, although impractical, is an approximation of the upper bound of any 4×4 1-bpp algorithm.

The codecs in Figure 5 and 6 all show increased quality with increasing mip sizes; this reflects the fact that larger images typically have more inter-pixel correlation than subsampled versions of the same images. DXT3 does not exploit inter-pixel correlation and hence does not benefit from increasing mip sizes as will be seen later in Figure 8.

Figure 7 shows screen captures from Oblivion with low-bitrate alpha textures in use, demonstrating the effect of the compression on the final rendered image.

The results seen in the figure indicate that although we achieve very low PSNR for the alpha channel, the degradation in image quality in the resulting images is barely noticeable.

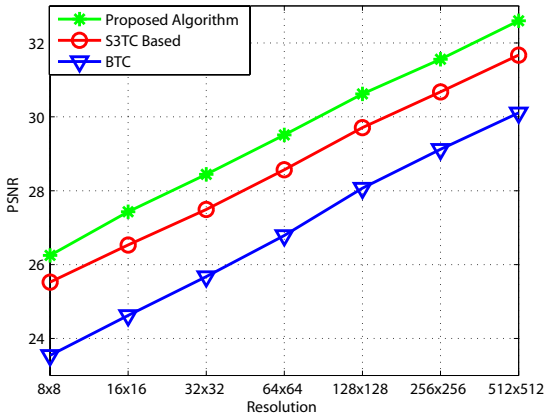

Figure 5: *Graph of our comparison of 32-bit compression modes. The PSNR scores shown are averages over 64 test images for seven different mip-levels.*

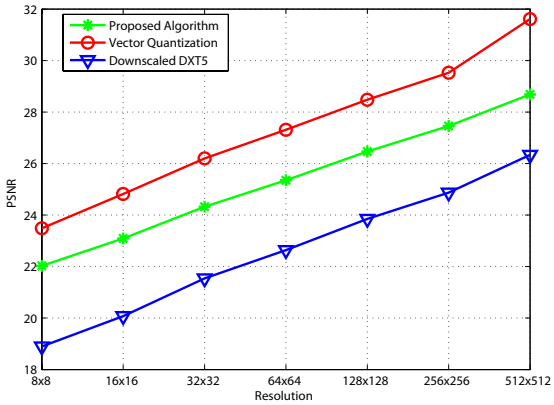

Figure 6: *Graph of our comparison of 16-bit compression modes. The PSNR scores shown are averages over 64 test images for seven different mip-levels. Note that the vector quantization system in this diagram is not feasible for texture compression, but should be regarded as an approximate upper bound on quality for a 4×4 1 bpp system..*

4. Evaluation of Table-based Compression

Since our low-bit rate algorithms seem competitive, we wanted to study whether the method of using base values and offset tables would be efficient also for 4-bpp data rates. In the next section, we therefore present a 4-bpp version of our algorithm, which is directly comparable to DXT5. However, first we discuss some interesting characteristics of that codec.

4.1. Motivation

One problem with the coding used in DXTC comes from the interesting fact that DXT5 in some cases allows higher accuracy than 8 bits. Setting the base values to 150 and 151,

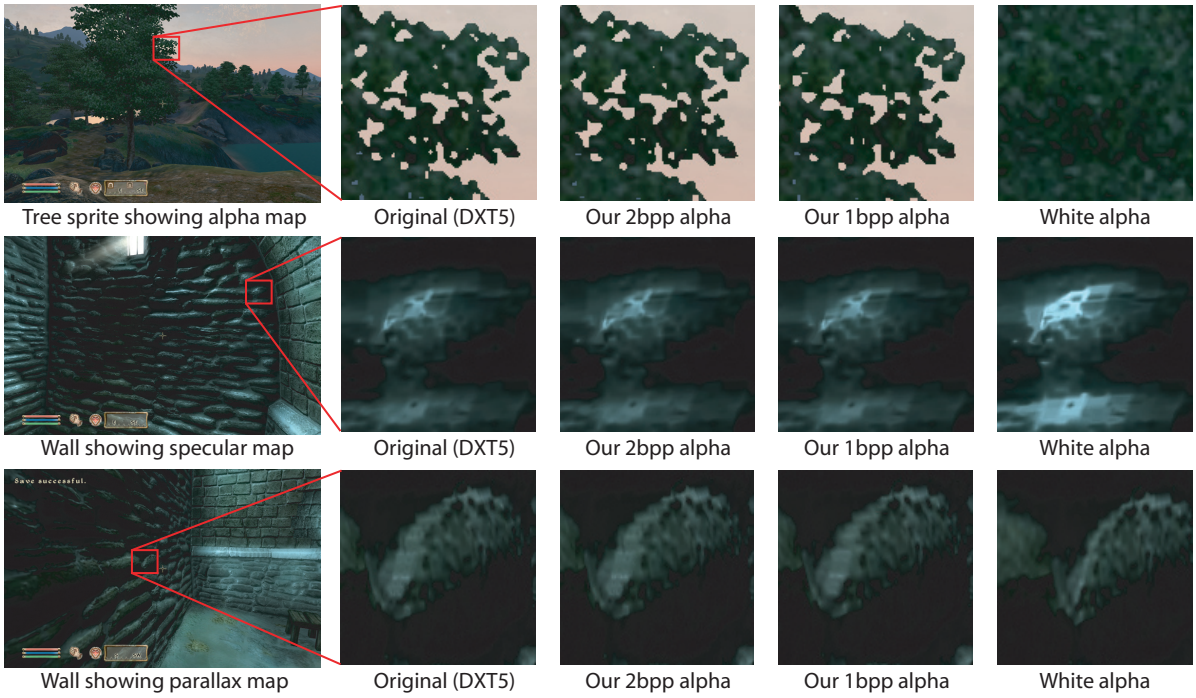

Figure 7: *Details from the low-bitrate comparison. The zoomed in images are, from left to right: Using the original DXT5 alpha channel, using our 2bpp algorithm, using our 1bpp algorithm, and using a completely white alpha channel.*

for instance, means that six shades of alpha between 150 and 151 are possible to represent. Thus DXT5 is almost as accurate as an 11-bit representation for very slowly varying blocks. This can be exploited when the original data is more than 8 bpp, such as if floating point alpha maps are used. However, when the original data is 8 bpp, this feature becomes a drawback. Assume the original block contains data in the range of 150-154. Selecting the base values 150 and 154, for instance, leads to the possible alpha values 150, 151, 151, 152, 152, 153, 153 and 154 after rounding. This will result in lossless (exact) compression, but there is no use for having two items with the value of 151: Selecting 150 and 157 as base values works just as well, also producing 150, 151, 152, 153 and 154, (as well as 155, 156, and 157). Seven out of 256, about 3%, of all possible base value combinations are redundant in this way, producing duplicate entries. When instead using tables, it is possible to make sure that no two table entries are the same, and it is thus possible to do better than DXT5 for 8 bpp data.

A second, more subtle, problem is with base values spaced further apart. In this case, slightly varying the two base values, and thereby the endpoints of the encoded interval, barely changes the intermediate interpolated values. Because of this, there are large amounts of base value choices that will result in similar final alpha values. It would be desirable to allow for a different distribution of intermediate values for a given set of endpoints.

We have investigated swapping out the DXTC approach of storing two base colors and interpolating between them for the table-based approach, using an 8-bit base value and an 8-bit offset table index.

4.2. Reduced table size

One issue with our 8-bit offset table index is that it can select between 256 different offset tables, each containing eight entries requiring 9-bits each. Storing these tables as-is on chip would require over 2 kb of storage, which may be prohibitive. In order to reduce this, we have opted to store just 16 tables with four 6-bit values, fitting in just 48 bytes of ROM. The four values in each table are copied, their signs changed and the value of one is added to provide the other four values in the table. We also treat the first four bits of the table index as a multiplier, multiplying them with the values in the table (selected by the other four bits) to produce the final eight offset values. The 16 stored base offset tables are shown in Table 2.

4.3. Results

The results of our comparison are shown in Figure 8. As can be seen, the proposed algorithm performs about 1 dB better

index	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	-17	-8	-5	-2	3	6	9	18
1	-12	-9	-6	-2	3	7	10	13
2	-12	-7	-4	-1	2	5	8	13
3	-12	-5	-3	-1	2	4	6	13
4	-11	-7	-5	-2	3	6	8	12
5	-10	-8	-6	-2	3	7	9	11
6	-10	-7	-6	-3	4	7	8	11
7	-10	-7	-4	-2	3	5	8	11
8	-9	-7	-5	-1	2	6	8	10
9	-9	-7	-4	-1	2	5	8	10
10	-9	-7	-3	-1	2	4	8	10
11	-9	-6	-4	-1	2	5	7	10
12	-9	-6	-3	-1	2	4	7	10
13	-9	-2	-1	0	1	2	3	10
14	-8	-7	-5	-3	4	6	8	9
15	-8	-6	-4	-2	3	5	7	9

Table 2: *offsets m_0 to m_7 for the 16 offset tables for 4 bpp compression. An index of 15 and a multiplier of 4 would result in the final offset table $\{-32,-24,-16,-8,12,20,28,36\}$. Note that the four rightmost columns are calculated as the inverse of the earlier columns with one added.*

on average than the one used in DXT5 on our set of test images. Again, exhaustive (optimal) compression was used for DXT5. The difference is most pronounced in tiles with widely varying values, where our method gains an advantage by being able to select from different distributions within a certain interval. An example of this is shown in Figure 9. DXT5 manages better on a few images in the test, in cases with small variations in alpha values.

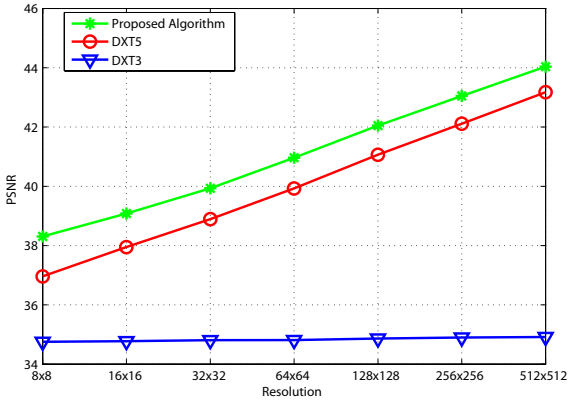

Figure 8: *Graph of our comparison of 64-bit compression techniques. The PSNR scores shown are averages over 64 test images for seven different mip-levels.*

5. Hardware Implementation Analysis

This section presents a detailed description of how decoding may be done in hardware for the 16-bit algorithm. We have not done a VHDL implementation of our algorithm, so we cannot say exactly how the hardware complexity compares to, for instance, that of DXT5. However, it is fair to assume that it is higher; in DXT5 the three-bit indices are stored directly, whereas in the proposed algorithm they must be decoded, which adds to complexity. Still, we hope that the huge savings in memory bandwidth and storage will more than make up for this added complexity, especially over time if growth in computation continues to outpace that of memory bandwidth.

Figure 9: *Differences between DXT5 coding and the proposed algorithm on a hand-picked example, where the large range of values gives DXT5 difficulties.*

The proposed hardware implementation is divided into two steps; the top row in Figure 10 (*i* through *vi*) describes logic that is shared between texels in the block, whereas the bottom row (*vii* through *ix*) shows per-texel logic. For instance, if four texels from the same block are to be decoded for a bilinear blend, one copy of the upper row logic and four copies of the lower row logic would be sufficient to decode all four texels in parallel.

Assume that we want to decode the pixel marked with a circle in Figure 10(a) where the coordinates are $(x_1,y_1) = (2,0)$, $(x_2,y_2) = (1,3)$. The job of the per-block logic is to draw the line between these two coordinates. First it is checked that $4y_1 + x_1 \leq 4y_2 + x_2$, if this is not the case the points are swapped. This is implemented using one comparator and two multiplexors as shown in Figure 10(*i*). Figure 10(b) shows the possible lines from (x_1,y_1) (red pixel) to (x_2,y_2) after sorting. The next step mirrors the coordinates for lines with negative Δxs (marked with green in Figure 10(b)) using two adders and a multiplexor, as show in Figure 10(*ii*). Figure 10(c) shows the possible lines after mirroring. The following step transposes the coordinates if they are below the diagonal (marked with green in Figure 10(c)) using five adders and a multiplexor (Figure 10(*iii*)). After transposition, the possible lines are shown in Figure 10(d). The correct line pattern can now be selected with a multiplexor selecting from ten different 10-bit vectors as shown in Figure 10(*iv*). The possible patterns are shown in Figure 10(e). Next, the bit sequence is transposed back. This is implemented in Figure 10(*v*) using twelve one-bit multiplexors. The numbering of the bits follows that of Figure 10(f). The result for our example vector is shown in Figure 10(g). The last step in the per-block logic is to mirror the bit sequence using 16 multiplexors. The bit pattern (shown

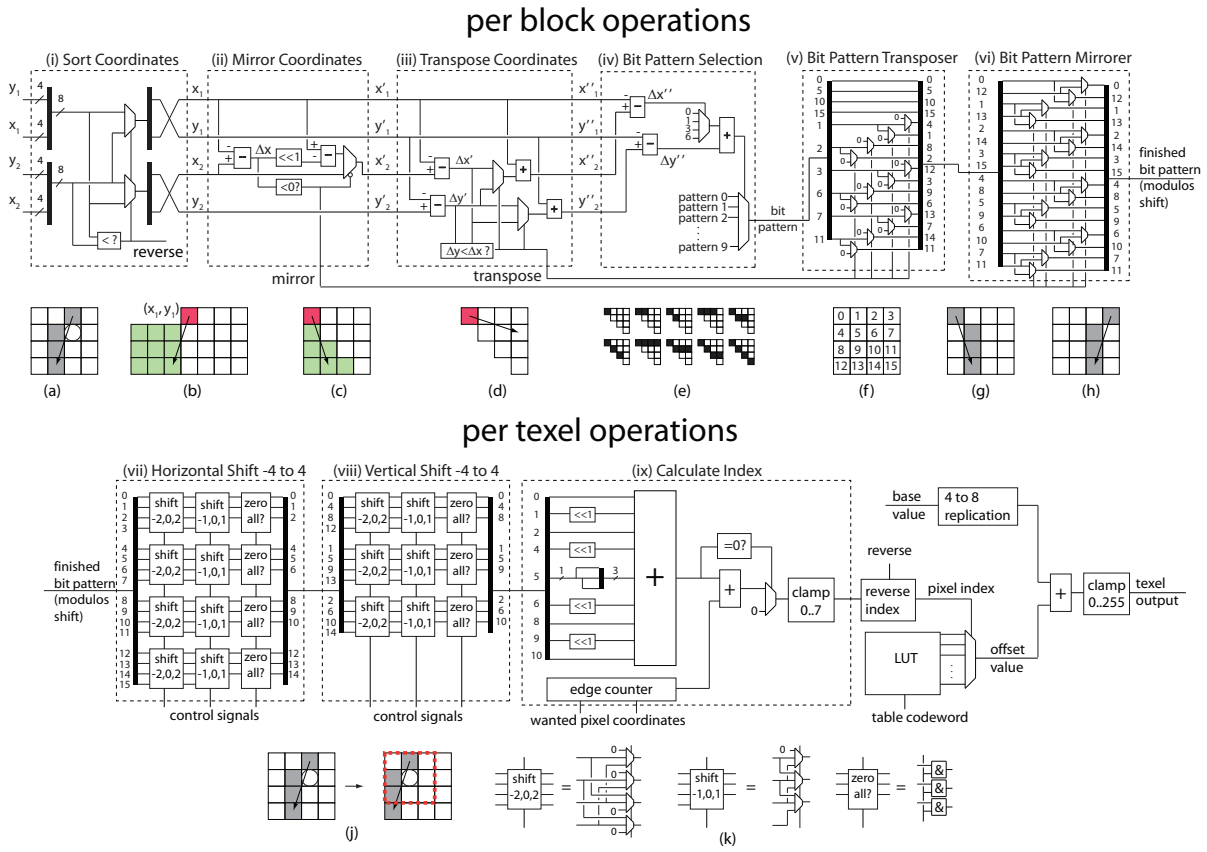

Figure 10: *Hardware layout diagram.*

in Figure 10(h)) is now correct up to shifting in the x- and y-directions.

The first step in the per-texel logic is to shift the bit pattern. The shifting is for two purposes; the first is to make sure that (x_1, y_1) and (x_2, y_2) end up in the correct positions as shown in Figure 10(a), and in our example one left shift and zero vertical shifts are needed. The second purpose is to place the texel to be decoded in the middle of the window marked with a red dashed rectangle in Figure 10(j). Another left shift is needed for this. The two types of shifts are naturally combined and carried out in hardware blocks (*vii*) (horizontal shift) and (*viii*) (vertical shift) in Figure 10. The horizontal shift logic uses four units that can shift two bits and four units that can shift one bit. This way all shifts from -3 to 3 can be encompassed. For four-step shifts, the output is always zero, so the last column in Figure 10(*vii*) implements a conditional zeroing of all bits. Each shift unit is made out of three or four three-way multiplexors, as can be seen in Figure 10(k), and each conditional zero-all circuit consists of three AND gates. Thus the two shifting stages can in total be implemented using 49 multiplexors and 21 AND gates.

The output from the shifters is the neighborhood of the pixel of interest marked with the dashed red rectangle in Figure 10(j). The components are now summed together in step (*ix*); a 4-neighbor is worth 2, a diagonal neighbor 1, the pixel itself is worth 7 and if this sum is larger than zero, edges also count. The sum is clamped to the interval $[0,7]$ and the resulting pixel index is reversed (i.e., an index k is replaced by $7-k$) if the reverse flag from stage (*i*) is set. A look-up table finds the correct table of eight values using the table codeword, and the recently calculated pixel index is used to select one of these eight values. Finally, the base value is expanded to eight bits and added; the result is clamped to the interval $[0,255]$ and the final texel output is thereby obtained.

Note that steps (*ii*) through (*vi*) could be replaced by a look-up table; there are exactly 136 different line patterns, and since each pattern consists of 16 bits this would mean a ROM of 272 bytes. An alternative would be to store the finished 3-bit indices in ROM — 48 bits would then be needed per pattern, yielding 816 bytes. The entire chain of steps (*ii*) through (*ix*) would then be replaced. Such an implementation would certainly be simpler to implement and debug, but we believe it would be more costly in terms of gate count.

6. Conclusions and Future work

We have introduced two low-bitrate alpha compression algorithms, based on the base value and offset table representation introduced in ETC, compressing the alpha channel of a texture down to 32 or 16 bits per 4×4 block. Further, we have investigated the performance of these algorithms, and their impact on images rendered using textures compressed with them.

A straight comparison of compression performance of our 1 bpp algorithm has been difficult, since there is very little literature on such low-bit rate algorithms. We have shown, however, that our 2 bpp algorithm achieves better PSNR scores than BTC, the precursor to DXT5. We have also shown that our 2 bpp and 4 bpp algorithms that use a base value- and offset table pair also leads to better compression than the interpolation approach used in DXTC.

A very intriguing finding is how little visual impact high compression of alpha channels had in our application investigation. Although our study is very limited, it indicates that compressing alpha channels used to store specular or parallax maps down to one bit per pixel can lead to rendered images virtually indistinguishable from those rendered using DXT5-compressed alpha channels, despite a 75% reduction in bandwidth used for that channel. When compressing an alpha channel used for the opacity information of a sprite such as in the top row of Figure 7, the difference is much more pronounced, a result of thresholding used in the shader. Despite this, in the particular example in Figure 7 it is not clear that the resulting image is subjectively worse, and we believe that our algorithm would be a viable alternative in general for such opacity data. Even so, there will of course always be cases where sacrificing the quality of the alpha channel is simply not acceptable.

One possible drawback of using these low-bitrate alpha compression modes is that, if used for RGBA textures, the bits saved in the alpha channel might lead to impractical bit sizes for the compressed texture blocks. In DXT5, 64 bits are spent on the RGB part and 64 bits on the alpha part. If we go down from 64 bits to 32 bits for the alpha part we end up with 96 bits per block, which is not a power of two and hence not burst-friendly. In such a case, we would suggest spending the saved bits in the color channel instead. Alternatively, reducing the bit count for the RGB part from 64 to 48 might be attractive if combined with the 16 bit alpha mode, resulting in an attractive burst size of 64 bits. However, we make no explicit suggestions for new RGBA texture compression formats, instead leaving that for future work. Note however that using the 16- and 32 bit modes to compress one- or two-channel textures such as alpha-only or luminance alpha textures will of course give attractive burst sizes that are powers of two.

It should be noted that, although designed with mobile applications is mind, the proposed methods are certainly not limited to such use; games on PCs and consoles can also be bandwidth limited, and such devices would hence also benefit from the proposed schemes.

Finally, all compression modes currently included in the DirectX standard assume correlation between the three color channels. For applications where there is no such correlation, it would be interesting to investigate compression modes which apply some of the proposed algorithms on each channel individually, which should lead to higher quality results or higher compression ratios in these special cases.

References

[AMN03] AILA T., MIETTINEN V., NORDLUND P.: Delay Streams for Graphics Hardware. *ACM Transactions on Graphics, 22*, 3 (2003), 792–800.

[AMS03] AKENINE-MÖLLER T., STRÖM J.: Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics, 22*, 3 (2003), 801–808.

[ATI05] ATI: *Radeon X800: 3Dc White Paper*. Tech. rep., 2005.

[BAC96] BEERS A., AGRAWALA M., CHADDA N.: Rendering from Compressed Textures. In *Proceedings of SIGGRAPH* (1996), pp. 373–378.

[CDF*86] CAMPBELL G., DEFANTI T. A., FREDERIKSEN J., JOYCE S. A., LESKE L. A., LINDBERG J. A., SANDIN D. J.: Two Bit/Pixel Full Color Encoding. In *Proceedings of SIGGRAPH* (1986), vol. 22, pp. 215–223.

[DM79] DELP E., MITCHELL O.: Image Compression using Block Truncation Coding. *IEEE Transactions on Communications 2*, 9 (1979), 1335–1342.

[Fen03] FENNEY S.: Texture Compression using Low-Frequency Signal Modulation. In *Graphics Hardware* (2003), ACM Press, pp. 84–91.

[INH99] IOURCHA K., NAYAK K., HONG Z.: System and Method for Fixed-Rate Block-based Image Compression with Inferred Pixels Values. In *US Patent 5,956,431* (1999).

[KSKS96] KNITTEL G., SCHILLING A., KUGLER A., STRASSER W.: Hardware for Superior Texture Performance. *Computers & Graphics 20*, 4 (July 1996), 475–481.

[Owe05] OWENS J.: Streaming Architectures and Technology Trends. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 457–470.

[SAM04] STRÖM J., AKENINE-MÖLLER T.: PACKMAN: Texture Compression for Mobile Phones. *In Sketches program at SIGGRAPH,* (2004).

[SAM05] STRÖM J., AKENINE-MÖLLER T.: iPACKMAN: High Quality, Low Complexity Texture Compression for Mobile Phones. In *Graphics Hardware* (2005), ACM Press, pp. 63–70.

[TK96] TORBORG J., KAJIYA J. T.: Talisman: commodity realtime 3D graphics for the PC. In *International Conference on Computer Graphics and Interactive Techniques* (1996), ACM Press, pp. 353–363.